Infrastructure for Component-Based DDS Application Development [*]

William R. Otte, Aniruddha Gokhale, and Douglas C. Schmidt
Dept of EECS, Vanderbilt University
{wotte,gokhale,schmidt}@dre.vanderbilt.edu

Johnny Willemsen
Remedy IT
jwillemsen@remedy.nl

Abstract

Enterprise distributed real-time and embedded (DRE) systems are increasingly being developed with the use of component-based software techniques. Unfortunately, commonly used component middleware platforms provide limited support for event-based publish/subscribe (pub/sub) mechanisms that meet both quality-of-service (QoS) and configurability requirements of DRE systems. On the other hand, although pub/sub technologies, such as OMG Data Distribution Service (DDS), support a wide range of QoS settings, the level of abstraction they provide make it hard to configure them due to the significant source-level configuration that must be hard-coded at compile time or tailored at run-time using proprietary, ad hoc configuration logic. Moreover, developers of applications using native pub/sub technologies must write large amounts of boilerplate "glue" code to support run-time configuration of QoS properties, which is tedious and error-prone. This paper describes a novel, generative approach that combines the strengths of QoS-enabled pub/sub middleware with component-based middleware technologies. In particular, this paper describes the design and implementation of DDS4CIAO which addresses a number of inherent and accidental complexities in the DDS4CCM standard. DDS4CIAO simplifies the development, deployment, and configuration of component-based DRE systems that leverage DDS's powerful QoS capabilities by provisioning DDS QoS policy settings and simplifying the development of DDS applications.

Categories and Subject Descriptors C.4 [*Computer Systems Organization*]: Performance of Systems—performance attributes; C.2.4 [*Software Engineering*]: Distributed Systems—components, deployment; D.2.11 [*Software Engineering*]: Software Architectures—domain-specific architectures

General Terms Software, Components, Deployment, Optimizations

[*] This work was supported in part by NSF CAREER 0845789 and CNS 0915976, and a contract from Northrop Grumman and AFRL GUTS. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, AFRL, or NGC.

Keywords component-based real-time systems, predictable deployment

1. Introduction

The trend towards realizing enterprise distributed real-time and embedded (DRE) systems motivates the use of component-based middleware, such as the OMG's Lightweight CORBA Component Model (LwCCM) [11]. Component-based middleware offers DRE system developers significant flexibility in modularizing their system functionalities into reusable units, simplifies the deployment and configuration of the systems, and supports dynamic adaptation of system capabilities. Deployment and configuration standards, such as the OMG's Deployment and Configuration (D&C) specification [14], play a major role in realizing these capabilities.

Existing and planned enterprise DRE systems must increasingly support large data spaces generated by thousands of collaborating nodes, sensors, and actuators that must exchange information to detect changes in the operational environment, make sense of that information, and effect changes. These capabilities require scalable publish/subscribe (pub/sub) semantics [6] that support a range of QoS properties, that control properties, such as liveliness, latency, deadlines, timing, and reliability. Unfortunately, the conventional component technologies used to develop enterprise DRE systems either do not provide first class support for pub/sub semantics or do so in an ineffective manner that is not scalable and does not support real-time QoS properties.

A standardized, QoS-enabled pub/sub technology called the OMG Data Distribution Service (DDS) [12] has emerged as a promising pub/sub technology to support the requirements of enterprise DRE systems. DDS includes standard QoS policies and mechanisms to handle data (de)marshaling, node discovery and connection, and configuration. Middleware based on the DDS standard has been applied successfully in mission-critical domains, such as air traffic management systems [5] and tactical information systems [7].

While the DDS specification simplifies key implementation aspects of pub/sub application, these benefits come at price of increased complexity of configuration glue code that must be written and maintained. Moreover, this configuration boilerplate code tightly couples the QoS configuration of a DDS application at compile-time, unless application developers create ad hoc methods of specifying the middleware configuration at run-time. Analysis [1] has shown that as 80% percent of DDS-related code in a typical applications is associated with configuring the middleware. Likewise, over half of the DDS API that developers must learn is configuration-related.

Addressing these deployment and configuration requirements of modern DRE systems calls for component-based middleware, such as LwCCM, to provide first-class support for QoS-enabled, pub/-sub technologies, such as DDS. This need has been recognized and documented through the efforts of industry and academic collaborators in the OMG *DDS for Lightweight CCM* (DDS4CCM) [13] specification. Implementing this specification is hard, however, due to inherent and accidental complexities in integrating LwCCM and DDS. The inherent complexities stem from (1) differences in the language bindings and memory management strategies of the two middleware technologies, (2) incompatibilities between the various specifications, (3) deployment and configuration challenges to recognize DDS abstractions within LwCCM, and supporting variants of DDS in a single LwCCM implementation. The accidental complexities stem from (1) manual approaches to creating the deployment and configuration metadata for DDS elements within LwCCM, and (2) the need to minimize run-time overhead imposed by both the deployment and configuration metadata, and the additional abstraction atop native DDS.

This paper describes how we have integrated LwCCM and DDS to address the inherent and accidental complexities described above as follows:

1. We make systematic use of the extensible interface pattern in the form of mixins to extend existing interfaces as well as the deployment and configuration metadata to bridge the incompatibilities between the two technologies.
2. We describe a template-driven code generation approach that maximizes the potential for portability between various DDS implementations and maximizes maintainability.
3. We provide options to customize the integration, which ensures that the runtime footprint of the resulting system does not pay unwanted memory footprint penalties.
4. We support improvements to the D&C approach mandated by the DDS4CCM specification.

Our contributions enable the realization of a product-line of DDS4CCM systems where it is possible to vary the implementations of the DDS technology used as well as support a wide range of port types for the LwCCM component technology. Empirical evaluations of our approach demonstrate that our implementation of the DDS4CCM specification, which we call DDS4CIAO, substantially eases the development of DDS-based applications while providing performance almost identical to native DDS applications.

The remainder of this paper is organized as follows. Section 2 summarizes key challenges encountered when integrating DDS within LwCCM; Section 3 describes the design of DDS4CIAO that resolves the challenges described in Section 2.3; Section 4 examines the code generation of DDS4CIAO and analyzes the results of experiments that evaluate the performance of DDS4-CIAO; Section 5 compares DDS4CIAO with related work, and Section 6 presents concluding remarks.

2. Impediments to Integrating LwCCM and DDS

In this section we present both the inherent and accidental challenges in providing first class support for Data Distribution Service (DDS) within the Lightweight CORBA Component Model (Lw-CCM).[1] To better appreciate these challenges, we first provide an overview of LwCCM and DDS, and the deployment and configuration standard. Subsequently we elaborate on the challenges.

[1] The LwCCM is a subset of the OMG CORBA Component Model. In the rest of this paper we refer to LwCCM because of our focus on DRE systems but the issues apply equally well to CCM.

2.1 Overview of Relevant Middleware Technologies

This section provides an overview of OMG LwCCM and OMG DDS.

2.1.1 The Lightweight CORBA Component Model (LwCCM)

The OMG Lightweight CCM (LwCCM) [11] specification standardizes the development, configuration, and deployment of component-based applications. LwCCM uses CORBA's distributed object computing model as its underlying architecture, so applications are not tied to any particular language or platform for their implementations. *Components* in LwCCM are the implementation entities that export a set of interfaces usable by conventional middleware clients as well as other components. Components can also express their intent to collaborate with other components by defining *ports*, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed messages with one or more components.

Homes are factories that shield clients from the details of component creation strategies and subsequent queries to locate component instances. A container in LwCCM provides an operating environment that can be configured and shared by components requiring a common set of QoS policies and functional support.

2.1.2 The OMG Deployment and Configuration

The OMG Deployment and Configuration (D&C) specification [14] provides standard interchange formats for metadata used throughout the component application development lifecycle, as well as runtime interfaces used for packaging and planning. Figure 1 depicts an architectural overview of the OMG D&C model.

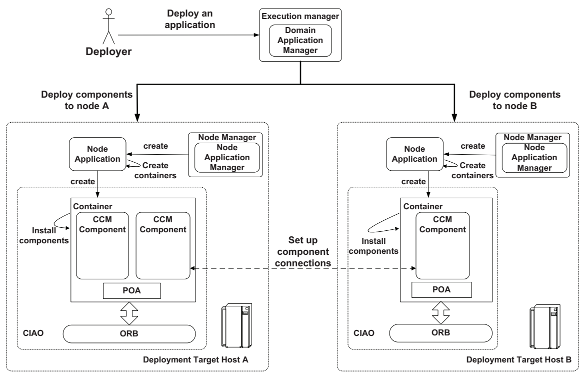

Figure 1. An Overview of OMG Deployment and Configuration Model

The runtime interfaces defined by the OMG D&C specification for deployment and configuration consists of the two-tier architecture comprising a set of global entities used to coordinate deployment and a set of node-level entities used to instantiate component instances and configure their connections and QoS properties. In addition to the runtime entities described above, the D&C specification also contains an extensive data model that is used to describe component applications throughout their deployment lifecycle. The D&C metadata defined by the data model contains a section where arbitrary configuration information may be included in the form of a sequence of name/value pairs, where the value may be an arbitrary data type. This configuration information is used to describe everything from basic configuration information (such as shared library entrypoints and component/container associations) to more complex configuration information (such as QoS properties or initialization of component attributes with user-defined data types).

2.1.3 Overview of the OMG Data Distribution Service (DDS)

The OMG DDS specification [12] defines a standard architecture for exchanging data in pub/sub systems. DDS provides a global data store in which publishers and subscribers write and read data, respectively. DDS provides flexibility and modular structure by decoupling: (1) *location*, via anonymous publish/subscribe, (2) *redundancy*, by allowing any numbers of readers and writers, (3) *time*, by providing asynchronous, time-independent data distribution, and (4) *platform*, by supporting a platform-independent model that can be mapped to different platform-specific models, such as C++ running on VxWorks or Java running on Real-time Linux.

DDS entities include *topics*, which describe the type of data to be written or read, *data readers*, which subscribe to the values or instances of particular topics, and *data writers*, which publish values or instances for particular topics. Moreover, *publishers* manage groups of data writers and *subscribers* manage groups of data readers.

Properties of these entities can be configured using combinations of DDS-supported QoS policies. Each QoS policy has $\sim$2 parameters, with the bulk of the parameters having a large number of possible values, *e.g.*, a parameter of type long or character string. DDS provides a wide range of QoS capabilities that can be configured to meet the needs of topic-based distributed systems with diverse QoS requirements. DDS' flexible configurability, however, requires careful management of interactions between various QoS policies so that the system behaves as expected. It is incumbent upon the developer to use the QoS policies appropriately and judiciously.

2.2 Addressing Limitations in the LwCCM Port System via DDS4CCM

The OMG's DDS4CCM [13] specification was developed to overcome the following limitations in LwCCM and DDS while still preserving the inherent advantages of each technology.

Limitation 1: Support for event-based pub/sub communication in LwCCM is extremely limited. LwCCM does not specify a particular distribution middleware that must be used inside the container for communicating events. While this approach allows a substantial amount of flexibility on the part of implementation authors, allowing them to choose to implement this support using, for example, the CORBA Event Service or CORBA Notification Service, has two important drawbacks. First, the integration of new pub/sub middleware requires modification of not only the core container implementation, but potentially also the deployment and configuration infrastructure in order to properly operate. As a result, this is an extremely complex task, often requiring that the integrator be an expert in both the LwCCM implementation and the desired distribution middleware.

Second, in order to remain completely generic, the interface available to component developers for event-based communication consists of only two operations: 1) a single method per port that allows for a single event to be published at a time, and 2) a single callback operation that provides an event to the component as it arrives. This prevents the component from taking advantage of many features of pub/sub messaging middleware that provide for status notifications and per-message QoS adjustment.

Limitation 2: Grouping of related services must be done in an ad-hoc manner. In many cases, services offered by a component require more than one interface in order to provide correct operation. As a simple example, consider a scenario in which two components expect to cooperate via mutually connected interfaces. In this scenario, one component provides an interface "A" and requires an interface "B", while another component provides complementary ports (*i.e.,* provides "B" but requires "A"). In order for semantically correct operation, the connections for both "A" and "B" must go to the same component, but there exists no way in LwCCM to indicate this constraint on an interface level. To accomplish this goal, developers must rely on ad-hoc naming conventions and documentation. This approach has the unfortunate side effect of complicating the planning process and potentially causing subtle and pernicious run-time errors if connections are mis-configured.

The DDS4CCM specification addresses these limitations by enabling LwCCM to leverage the powerful pub/sub mechanisms of DDS. First, it provides a substantially simplified API to the application developer that completely removes the configuration of the DDS middleware from the scope of the application developer. Second, it provides a set of ready-to-use ports that hide the complexity and groups data writing/access API with the appropriate callback and status interfaces. Third, by providing integration with the LwCCM container, DDS applications are now able to take advantage of robust and mature deployment and configuration technologies that obviates the need to write boilerplate application startup code, run-time configuration of QoS policies, and coordinated startup and teardown of applications across multiple nodes.

In particular, DDS4CCM proposes two new constructs — *extended ports*, which allow for the grouping of related services, and *connectors*, which allow for flexible integration of new distribution middleware. These new entities are defined using an extension of the IDL language for components (IDL3) called IDL3+. It is possible to map each of these new IDL3+ language constructs back to basic IDL3 using simple mapping rules to enable inseparability with older CCM implementations. Next, we provide a brief overview of these enhancements.

Extended Ports: Extended ports provide a mechanism whereby component designers can group semantically related ports to create coherent services offered by a component. These extended ports, defined using a new IDL keyword `porttype`, are defined outside the scope of components. Extended ports are allowed to contain any number of standard LwCCM ports in either direction. While these ports are allowed in terms of the specification to contain standard LwCCM event ports, in practice this is highly unlikely due to the limitations outlined earlier. Moreover, in combination with connectors (described next), these extended port definitions could be used to recreate the behavior of the existing standard CCM event infrastructure.

Listing 1 shows IDL for an example extended port. In this example, we create a service whereby one component may notify another of data that is ready to be sent, and the destination component may optionally choose to pull that data from the source component. Since each of the interfaces `Data_Source` and `Notifier` are semantically linked, *i.e.*, operation of the component application would be fundamentally broken if these ports are not pairwise connected, they are grouped into a single `porttype`. This is an indication to both high level modeling tools and the component run-time that these ports must be connected as a pair, and can generate appropriate deployment plan meta-data to connect them at run-time. Extended ports are assigned to components using two new IDL3+ keywords. The `port` keyword indicates that the component supports the extended port *as described*. The `mirrorport` keyword indicates that the component *inverts* the direction of the extended port, *i.e.*, facets become receptacles.

Listing 1. Extended Port IDL

```
interface Data_Source {
  Data pull (in long uuid);
};
```

```
interface Notifier {
  void data_ready (in long uuid);
};
porttype NotifiedData {
  provides Data_Source data_source;
  uses Notifier data_ready;
};
component Sender {
  port NotifiedData data_out;
};
component Receiver {
  mirrorport NotifiedData data_in;
};
```

Some extended ports may vary only in the data type used as parameters. In order to avoid the necessity of re-defining an extended port for each new data type, IDL3+ offers a new template syntax that may be used to define services that are generic with respect to data type.

Connectors: While the extended port feature described above is quite useful, their power is most suited to providing novel communications mechanisms to components that provide/use those interfaces. In order for the extended ports to provide a coherent interface to a new distribution middleware, such as DDS or the CORBA Event Service, the business logic that supports that abstraction must be contained in some entity. This unit of business logic is called a *connector*. Connectors combine one or more extended ports to provide well-defined interfaces to new distribution middleware or communication techniques between components. In many cases, a single connector will support at least two extended ports, one intended for each "side" of the communications channel. By separating the core communications business logic, these connectors can then be used as COTS components across several applications without requiring modification of the core container code.

Connectors are defined similar to a component, using the new IDL3+ keyword `connector`. Connectors may contain, of course, one or more extended ports. In addition, they may also support attributes which are intended to be used to assist in runtime configuration, *i.e.* topic names, port numbers, QoS parameters, *etc.*. Finally, connectors also support inheritance which can be used to extend existing connectors with new capabilities. At runtime, instead of creating a new IDL type structure for the connector infrastructure, they are defined as components, deriving their interface from the same `CCMObject` used by regular components. Indeed, in the IDL3+ to IDL3 mapping, the `connector` keyword becomes `component`. This approach is much desirable in that no additional work is necessary in the D&C toolchain to support the deployment and configuration of connectors. Moreover, connector implementations can take advantage of the same Component Implementation Framework that is available to standard LwCCM components and thus can take advantage of advances in services offered by the container.

2.3 Challenges in Integrating LwCCM and DDS

Although the DDS4CCM specification attempts to address the limitations of individual technologies, realizing an implementation of the DDS4CCM specification is fraught with multiple inherent and accidental complexities explained below:

Challenge 1: Indicating that a connector implementation has been fully configured, and should be made ready for execution. After a connector implementation has received all necessary configuration information, it must proceed to create the underlying low-level DDS entities (*e.g.*, `DomainParticipant`, `DataWriter` and/or `DataReader`) that are necessary for correct operation. To accomplish this task, the specification mandates the use of an operation called `configuration_complete` on the external connector interface. This operation, however, is not delegated to the connector business logic and thus is insufficient to fully inform the connector implementation of completed configuration. Section 3.1 discusses our approach to resolve this challenge.

Challenge 2: Reducing D&C-related runtime memory footprint. The DDS4CCM specification mandates the use of LwCCM Homes (which nominally act as factories for component instances) as the primary vehicle for passing configuration information from the deployment plan to individual connector implementation during deployment. While this approach is certainly functional and sound (and in keeping with the spirit of the LwCCM specification), our experience developing component applications with LwCCM reveals that the home entity often adds very little value to the configuration of individual component, or in this case connector, instances. In most cases, the home implementation is little more than a simple factory that directly instantiates the component and nothing else. Meanwhile, the home instance carries a non-negligible amount of runtime footprint due to the CORBA interface and accompanying home-specific generated container code that is necessary. Section 3.2 discusses our approach to resolve this challenge.

Challenge 3: Reducing Connector-related runtime memory footprint. The decision to treat connectors for all intents and purposes as full LwCCM components greatly simplifies the implementation by substantially reducing the number of changes in the core container necessary to support the specification. A consequence of this decision, however, is that the runtime footprint of a LwCCM application using connectors could substantially increase. For example, assuming a deployment where each component instance has an associated connector instance, the number of actual "components" in the deployment is doubled. In memory-constrained DRE systems, this can be a significant impediment. Section 3.3 discusses our approach to resolve this challenge.

Challenge 4: Supporting Local Interfaces as Facets All of the extended ports contained in the DDS4CCM specification are defined as "local interfaces". Local interfaces are significantly different from standard CORBA interfaces due to the fact that they are not generated with any of the infrastructure necessary to support remote invocation. As a result, any invocation on these interfaces does not travel through the CORBA internal infrastructure and as such only incurs overhead nominally involved in a virtual method invocation. The problem this strategy causes with the deployment and configuration aspect of LwCCM is very subtle: since these local interfaces lack the necessary remoting code, it is impossible to pass references to these local objects through a standard CORBA interface. Indeed, this behavior is undefined; any attempt to do so will fail and cause an exception to be propagated to the caller. Unfortunately, all of the standard-defined connection methods, including the Component Navigation interfaces used by the D&C tooling to make connections between components rely on being able to retrieve object references to Facets over a standard CORBA interface and pass these references to the receptacle component over a similar interface. Not having an object reference for the extended port implies that the existing D&C tooling cannot be leveraged in a straightforward manner. Section 3.4 discusses our approach to resolve this challenge.

Challenge 5: Supporting Multiple DDS Implementations One significant benefit of writing DDS applications using the DDS4CCM API is that it potentially makes it substantially easier to switch between various DDS implementations. Prior work [16] has shown that differences in the architecture between these different implementations cause them to have different strengths depending on the architecture of the application and hardware environment. Moreover, due to the proprietary nature of most DDS implementations and the different licensing requirements of each implementation, the ability to quickly and easily switch the targeted implementation would greatly facilitate the development of COTS DDS components. While it is currently possible to target multiple DDS implementations *at compile time* due to the presence of a standard API, subtle differences in the implementations of these APIs can make this difficult to accomplish. Ideally, any implementation of the DDS4CCM specification would be architected in such a way that the core business logic of the connector is shielded from the differences between DDS implementations. In addition, the connector architecture could make it possible to delay the choice of DDS implementation from compile time to deployment time. Section 3.5 discusses our approach to resolve this challenge.

Challenge 6: Making it easy for users to define their own connectors The DDS4CCM specification provides for two connector types that correspond to common DDS usage patterns. The first provides for a state transfer pattern, and is intended to connect "Observable" components that publish state to other "Observer" components that consume that state. The second provides for event transfer connecting supplier components to consumer components. These two connectors, however, are not intended to be the only ones that are supported in the context of the specification. To that end, two "base" connectors are provided that collect the various configuration meta-data as attributes. It is intended that users be able to define their own connectors that are better suited to their usage cases. To support this capability, the code generation techniques should be extensible such that it is easy for users to create their own connectors without having to modify the code generators. Section 3.6 discusses our approach to resolve this challenge.

3. Resolving LwCCM and DDS Integration Challenges in DDS4CIAO

This section describes how we resolved the challenges in integrating LwCCM with DDS described in Section 2.3 by presenting the architectural and design choices made for DDS4CIAO, which is our implementation of the DDS for Lightweight CCM specification outlined in Section 2.2.

3.1 Accurate Indication of Successful Connector Configuration

The central difficulty outlined in **Challenge 1** from Section 2.3 revolves around the final configuration stage of the D&C process. In this case, there lies a crucial phase before the application is "activated", but after it is fully configured. In this portion of the D&C process, the connector business logic must make themselves ready for execution by, for example, instantiating various DDS entities. In Figure 2, which shows the lifecycle stages that connectors and components go through, this is represented by the "Passive" state. Unfortunately, the LwCCM specification currently provides no mechanism to communicate to the connector that it has entered this state; the only notification that is received when the component/connector becomes passive is when the prior state was "Active". To understand the reason for this, it is best to have a grasp of the layout of connectors and components at runtime.

Instantiated connectors consist of two primary pieces. First, there is a "Servant", which consists of the external CORBA interface and connector-specific container code. The Servant has two primary parts to its interface: (1) operations common to all connectors which come from the LwCCM specification (called the `CCMObject` interface), and (2) operations that result from the ports specified in the IDL declaration of the connector. Second is the "Executor", which contains the actual business logic that implements the connector. Operations on this interface result from two sources: (1) specification-defined lifecycle operations (called the

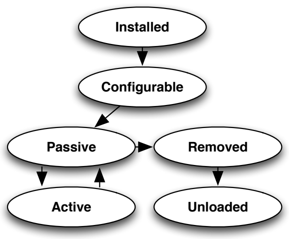

Figure 2. LwCCM Component and Connector Lifecycle Stages

`SessionComponent` interface), and (2) operations that result from the ports defined for the connector.

The `configuration_complete` operation mentioned in Section 2.3 is part of the `CCMObject` interface but is not, however, present on the `SessionComponent` interface so it cannot be directly delegated.[2] Unfortunately, the first lifecycle operation that is invoked on the Executor interface after its construction as defined by the LwCCM specification is `ccm_activate`. This lifecycle operation, however, must be disjoint from and occur later than `configuration_complete`.

One approach to work around this problem is to delay the creation of the DDS entities until the activation phase of the application lifecycle. This is problematic, however, because there exists no guarantee that a connector fragment will be activated *before* its connected component. If a component is activated before its connector and attempts to initiate outbound communication, that communication would naturally fail, potentially causing pernicious and difficult to reproduce errors. The ability for component business logic to receive a notification upon configuration completion but before activation has proven to be useful for components as well as connectors because connectors are anyway treated as components.

As a result, we have created a new interface that may be optionally used to extend the behavior of component executors to be able to receive these notifications. This interface, which we call `ConfigurableComponent`, uses a variation of the extension interface pattern to avoid changing the standard-defined `SessionComponent` interface. This new interface is intended to act as a mixin so that the component implementations wishing to receive `configuration_complete` will inherit from this in addition to the standard `SessionComponent` interface. The container, then, when it receives `configuration_complete` from the D&C tooling, will attempt a dynamic cast on the component implementation to determine if the operation should be delegated on a per-component basis.

3.2 Avoiding D&C-related Memory Footprint

Challenge 2, described in Section 2.3, deals with eliminating unnecessary footprint from the specification-defined deployment and configuration requirements of connectors. DDS4CCM connectors are configured via attributes present in the IDL interfaces defined by the specification, which allow for the fragment to be associated with a particular DDS domain and topic as well as the QoS policies.

Many hardware platforms commonly used for DRE systems remain extremely memory-constrained, so the additional run-time memory footprint imposed by the CCM home is at best undesirable. To avoid this additional overhead, DDS4CIAO provides the capability to install "un-homed" components and connectors. These unhomed components are allocated from simple factory functions exported from their implementation libraries in much the same manner that Homes are already constructed. Component-specific con-

[2] This artifact results from the standards specification.

tainer code, which is generated automatically from IDL, is then able to interpret the D&C plan meta-data and individually invoke the attribute setter methods on the component.

3.3 Reducing Connector-Related Memory Footprint

The solution **Challenge 3**, described in Section 2.3, attempts to reduce the runtime footprint of connector implementations. In order to accomplish this goal, we must determine which, if any services that a component requires that are not necessary for connector implementations. Given the limitations of the standard LwCCM event ports described in Section 2.2, it is highly unlikely that these inflexible port types would be used in the context of a connector — indeed, the extended port/connector infrastructure could be used to fabricate replacement infrastructure. Moreover, the DDS4CCM specification makes no use of the existing event infrastructure, making it an apt candidate for removal.

As a result, we sought to remove the event infrastructure from the connector infrastructure in such a way that it would still be present for standard components that may need to interface with legacy systems. In this case, there are two pieces to the event support in DDS4CIAO: (1) the base classes that provide support to the component-specific generated container code, and (2) the component-specific generated container code itself, which includes a component-specific context that provides services to the component business logic. The first portion of the event support — the base classes described above were split into two pieces — a *connector* base and a *component* base. The container base contains all necessary functionality for component and connectors minus the LwCCM event support. The necessary plumbing LwCCM event support is contained in the component base, which derives from the connector base. This way the code generation infrastructure can choose to omit support for the event infrastructure if desired by selecting a different base class for the generated code. Our approach makes this artifact configurable.

3.4 Supporting Local Facets

The solution to **Challenge 4** outlined in Section 2.3 is threefold. First, and most obviously, the Navigation and Introspection implementations generated for components with local facets and receptacles had to be modified to suppress any knowledge of these local ports. While this approach solves the issue of undefined behavior from trying to marshal one of these local object references, it also completely removes any standards-based mechanism by which a connection can be made by either the D&C tooling or any user attempting to use the Navigation interfaces. To address this undesired effect, a new connection API was created in the private interface to the CIAO container (which is our LwCCM implementation) that is used directly by the D&C tooling. This API accepts as arguments the string identifiers of two component endpoints as well as port names, and is able to use these to obtain references to the local Executor objects directly and create a connection without needing to marshal any local references over standard interfaces.

In order to make use of this new API, however, the D&C tooling needs an annotation on the connection meta-data so that it can be made aware that it should not attempt to use the standard Navigation API to make the connection. The data structure in the deployment plan that contains connection information encodes the type of connection (*e.g.*, Facet vs. Receptacle) as an enumerated value. While this enumeration could be extended to identify a new connection type (*i.e.*, LocalFacet), we endeavored to minimize changes to specification-defined types. The connection data structure does contain a section where requirements for deployments can be described using name/value pairs. This section would ordinarily be used to enumerate hardware capabilities or resources required by the connection. In this case, we require that any local facet connected be annotated with a requirement on the container, namely that it provide support for local facets — when the D&C tooling encounters this annotation it assumes the connection to be local.

3.5 Ensuring Portability of DDS4CIAO Implementation

As described in **Challenge 5** from Section 2.3, we would like to ensure that the design of the infrastructure is maximally portable in order to easily support implementations from multiple DDS vendors. This goal is complicated by the fact that despite the presence of a standard C++ language mapping, there are subtle and pernicious differences between the actual implementations of these mappings. Moreover, there exist also subtle behavioral differences between implementations that complicate source-level compatibility, *i.e.*, generated type-specific constructs such as `DataWriters` and `DataReaders` may have different namespaces and naming conventions, and indeed the same may be true of the entire API.

We addressed this challenge by using three approaches. The first approach targets the API that we wrote the implementations of the DDS4CCM basic ports against. The DDS specification, in addition to the widely supported C/C++ language binding, also has a language binding that maps the API into IDL interface definitions. This language binding is not widely implemented, but provides a promising vehicle for implementing portable DDS business logic in the context of the DDS4CCM basic ports. Since we are using the same IDL code generator as with the rest of the CIAO infrastructure, we can ensure that the APIs we are using to implement these ports are consistent.

Much of the work for supporting different DDS implementations then can be accomplished by providing an implementation of this IDL language binding. At first glance, this may seem a daunting proposition — however, this binding consists of only about 36 interfaces, many of whose functions may be directly delegated to the native implementation. The remaining problem with using this IDL-based approach is reconciling the differences between the CORBA types that are part of the IDL language mapping and the data types used natively by the DDS implementation. While this conversion could be handled inside the vendor-specific implementation of the IDL language binding, this approach would incur potentially expensive data copies. Fortunately, many DDS implementations provide a CORBA compatibility layer that allows them to directly use types generated by the IDL compiler.

3.6 Connector Code Generation

Generating code for user-defined connectors is the focus of **Challenge 6** from Section 2.3. Our experience developing code generators for our CORBA and LwCCM implementations has shown us that it is eminently undesirable to embed large amounts of business logic in generated code. This is largely due to the difficulty of maintaining and extending the code generators themselves. If there is a bug, modification, or extension to be made, this effort often involves at least two engineers — one who is familiar with the middleware or problem at hand, and another who is familiar with the process of extending and modifying the code generator. In addition to the extra personnel requirements, it often substantially increases the amount of time to test these changes, as not only does the initial proposed modification needs to be be tested (typically supplied to the code generation engineer as a handcrafted generated file), but also the final changes to the code generator and resulting modified output. For the same reason, this accidental complexity of the code generation process impedes the ability of users to create their own DDS4CCM connectors.

In order to avoid these accidental complexities, we designed the code generation infrastructure from the outset to contain zero DDS-4CCM business logic and to be extensible **without** the need to modify the code generator to add new connector implementations. The first, and most obvious step given the presence of parameterized modules from Section 2.2, was to leverage C++ templates for the implementations of the basic and extended DDS4CCM port types. Using C++ templates in this case allowed us to make generic two very important parts of the implementation — first, the core DDS-4CCM business logic contained in the basic and extended DDS-4CCM ports, but also the IDL wrapper (described in Section 3.5) around our target DDS implementation. These IDL wrappers require access to type-specific DDS entities (*e.g.* `DataWriters` and `Data Readers`) that are created by the code generation infrastructure that is part of the DDS implementation itself.

Connector implementations, then, are really a collection of template instantiations for the various basic and extended ports that are contained in their interface definition along with some configuration glue code. While we could certainly generate the source code for these connector implementations, that would still represent an obstacle to novel connector creation. Connectors themselves may contain a nontrivial amount of configuration business logic that interprets the values of attributes on the connector interface. As a result, if a user were to define a new connector with new configuration attributes, they would be required to modify the code generator to be able to use their new connector.

To address this concern, we elected to make connector implementations template classes as well. This allows the code generator for DDS4CCM to be extremely simple. In effect, the result of the code generation process is a header file that contains a set of C++ traits [10] which specify the properties necessary to use a particular IDL data type. These properties largely consist of the names of type-specific entities that are generated from the DDS infrastructure. These traits are then used to create concrete template instantiations of any required connector implementations. By default, we generate instantiations of the standard DDS4CCM connectors — the State and Event connectors described in Section 2.3. If a user defines their own connector in IDL, the code generator emits an include of a header file whose name derives from the name of the connector in IDL, and a concrete instantiation of a template class whose name is similarly derived. While the user must then provide an implementation of this template class, this is substantially less effort than would be required to modify the code generator.

4. Experimental Results

This section outlines two key empirical observations of the DDS4-CIAO implementation described in Section 3 which cover two important goals outlined in Section 1. First, in Section 4.2, we quantify the impact that the code generation capabilities of DDS4CIAO have on the development and maintenance of DDS-enabled applications. Second, in Section 4.3, we characterize the overhead that DDS-enabled applications must pay in terms of latency when using the DDS4CIAO abstraction versus using the DDS API directly.

4.1 Experimental Scenario

All results described below were obtained using a simple "ping-pong" application. We chose a simple example since the business logic of the application is not important to evaluate the qualities of DDS4CIAO. Rather we are interested in understanding the overhead, if any, of the integration of LwCCM with DDS. In this application, an instance struct containing an octet sequence of a configured length and a sequence number would be written to the DDS data space by a "Sender". The instance would arrive at a "Receiver" entity, after which a new instance of the struct would be published on a separate topic with an identical sequence number but a zero length octet sequence. The "Sender", upon receipt of the second message, repeats the process with a new sequence number up to a specified number of iterations.

Two versions of this application were produced. The first uses the native C++ DDS API, with all customary error checking included. In the second version, the "Sender" and "Receiver" were each implemented as CIAO components and used DDS4CIAO to interface with the DDS middleware.

4.2 Evaluation of Code Generation

To evaluate the effectiveness of the code generation techniques described in Section 3.6, the implementation source files from the experimental scenario outlined in Section 4.1 were analyzed with the SLOCCount [15] tool. This is a program which counts physical Source Lines of Code (SLOC), and uses a number of heuristics to discard any whitespace and commenting present. For the purposes of this evaluation, only implementation source files were counted, discarding header files containing only class definitions. The reason for this is that header files for the DDS4CIAO implementation are largely generated automatically based on the class interfaces.

The results from this tool are summarized in Table 1. If only the total SLOC for the native programs and the component implementations are compared, DDS4CIAO shows only a nominal improvement over that of the native implementation. It is important to consider, however, that the DDS4CIAO implementation contains a large amount of generated class skeletons which are created from the IDL interface descriptions from the component automatically (SLOC for which is shown in the "DDS4CIAO Generated" column of the table). When these lines of code are subtracted from the total for the DDS4CIAO implementation, the improvement becomes substantially more dramatic. In the case of the Sender component, the improvement is on the order of 50%, and for the receiver the difference is an order of magnitude. The reason for this discrepancy is the Sender programs — both native and DDS4CIAO — contains a substantial amount of code in common to measure latencies and calculate/display results.

Table 1. Comparison of Source Lines of Code

Component	Native Lines	DDS4CIAO Total	DDS4CIAO Generated	DDS4CIAO Actual
Sender	643	560	211	349
Receiver	293	128	118	10

4.3 Evaluation of the Overhead of DDS4CIAO

To evaluate the overhead due to abstraction over the native DDS API introduced by the DDS4CIAO implementation, the experimental scenario described earlier in Section 4.1 was used to evaluate the latency performance using a recent commercial DDS implementation and DDS4CIAO 0.8.3. Each configuration was executed for 1,000 iterations each with payload sizes along powers 2, from 16 to 8192 bytes. Each experimental run was executed in two transport configurations: once using UDP and again using Shared Memory transport. The experimental testbed consisted of Dell Optiplex 755 computers, with an Intel E4400 CPU, 2GB of RAM, and gigabit network connections.

The results for the experimental runs with the UDP transport protocol are shown in Figure 3, which compares the average latency for each payload size, and Figure 4, which compares the minimum latency results for each payload size. These results show that for this transport protocol, the average latencies are nearly identical. Figure 5 shows the results from the experimental runs configured with the shared memory transport. This average latency result shows that the DDS4CIAO abstraction introduces approximately a four percent overhead over the native implementation for the shared memory transport. The best case results for the shared memory experiment are shown in Figure 6.

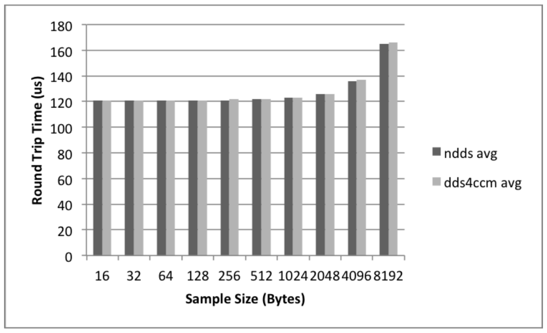

Figure 3. Ping Latency Average with UDP

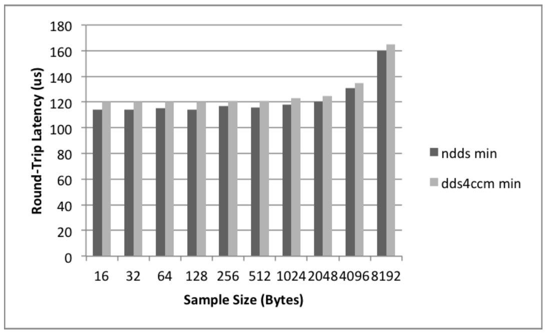

Figure 4. Ping Latency Minimum with UDP

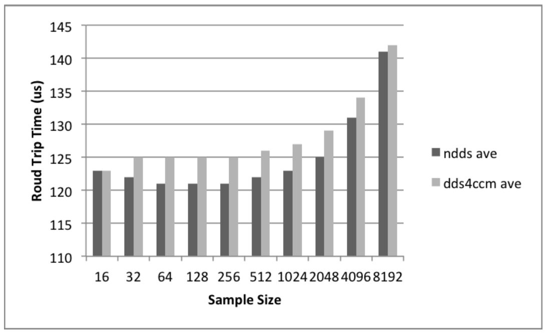

Figure 5. Ping Latency Average with Shared Memory

Table 2 summarizes the standard deviation of the experimental runs for both UDP and shared memory. These results show that the DDS4CIAO abstraction does not introduce additional jitter over the native implementation.

5. Related Work

This section compares our research on component-based DDS with related work.

PocoCapsule [8] is an Inversion of Control container based on the Dependency Injection (DI) design pattern. This component framework allows developers to use "Plain Old C++ Objects" (POCO) that have been decorated with PocoCapsule macros that allow the loading of these C++ classes into a PocoCapsule container. DDS4CCM and DDS4CIAO differ in several important aspects from PocoCapsule. First, DDS4CCM—and LwCCM in general—are industry standards that have language bindings defined for many programming languages. Second, PocoCapsule still requires some amount of low-level glue code in the component business

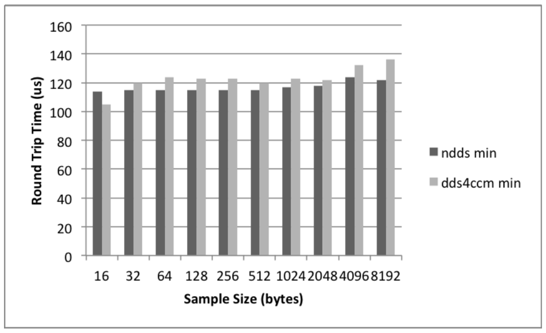

Figure 6. Ping Latency Minimum with Shared Memory

Table 2. Standard Deviation For All Experiments

Size	UDP	CIAO UDP	Shared	CIAO Shared
16	11.3	12.4	17.7	18.4
32	12.4	9.4	15	14.2
64	12.5	12.6	15.5	9.9
128	13.3	9.3	16	10.4
256	6.2	13.1	15.9	12.6
512	12.3	11.2	11.6	8.8
1024	14.7	8.1	15.7	12.1
2048	12.7	4.3	15.5	14.8
4096	7.1	13.7	15.3	10.8
8192	12.1	17.7	15.1	14.4

logic. Third, the DDS for PocoCapsule implementation currently only uses CORBA local interfaces to simulate small parts of the DDS API, and hence is not operable with standard-compliant DDS implementations.

Simple API for DDS (SimD) [2] uses C++ templates and template meta-programming to provide a simpler API for DDS that reduces the amount of infrastructure-related code required for DDS applications by an order of magnitude. Using SimD, a simple DDS application can be written in only 4 source hand-written lines of code, instead of dozens lines of code using the native API. While SimD reduces the complexity of the boilerplate code required for DDS applications, it differs substantially from DDS4CIAO in that it does not address run-time deployment and configuration capabilities provided by DDS4CIAO. Moreover, it has not yet been proposed as a standard.

Researchers at Real-Time Innovations, Inc [1] propose extensions to the DDS API to allow declarative configuration of DDS entities via an XML file that is interpreted at run-time. The application then queries the DDS middleware to obtain a particular `DataReader` or `DataWriter` that has been configured already with a domain and topic binding and QoS settings. While their work improves the state-of-the-practice in standards-based DDS application configuration, its capabilities are not as extensive as DDS4CCM and DDS4CIAO. First, our existing D&C tooling provides coordinated installation of application implementations and startup across multiple nodes. Second, the connector infrastructure developed for DDS4CIAO allows integration with other distribution middleware, such as CORBA, TENA, JMS, or even socket based network programs. Third, the decoupling provided by the DDS4CIAO implementation enables the selection of DDS implementation at deployment time.

SOFA [3, 4] is a component model with an integrated D&C framework that provides remote communication capabilities via a connector infrastructure similar in spirit to that which is part of the

DDS4CCM specification. SOFA, however, only provides connectors for CORBA and RMI distribution middleware. Our approach differs from that taken by SOFA in that the connectors implemented by DDS4CIAO are themselves lightweight components. The advantage of our approach is that any improvements to the QoS capabilities of the CIAO container can be automatically applied not only to all components deployed, but also connectors as well.

6. Concluding Remarks

This paper presents a novel generative approach for developing DDS-based component-oriented DRE systems. Our approach combines key advantages of the DDS middleware, such as low latency communication and extensive QoS policy support, with the strengths of a mature component model, such as simplified application composition and automatic deployment and configuration. We have prototyped and evaluated our approach via the DDS4CIAO middleware platform, which implements the Lightweight CCM (DDS4CCM) specification, while addressing a number of inherent and accidental complexities in integrating the DDS and LwCCM technologies. In particular, we have made extensive use of variants of the extensible interface pattern to extend the existing standard-defined LwCCM interface and deployment metadata to overcome incompatibilities between DDS and LwCCM and overcome oversights in the DDS4CCM specification. Additionally, we describe a template driven code generation technique that maximizes portability amongst DDS implementations while allowing users to extend DDS4CCM by defining their own connector types without having to modify the code generator.

Our experience developing applications with DDS4CIAO provided the basis for the following lessons learned:

Substantially reduced DDS application complexity. Tests and example applications developed with DDS4CIAO have shown that the simplified interface to the underlying DDS middleware, provided by the DDS4CCM specification, provides a platform that easier to write and develop DDS applications.

Automatic configuration of DDS middleware. By providing a strict separation of concerns between configuration-based aspects of DDS application development and configuration aspects, users can automatically configure the underlying middleware at deployment time using standards-based deployment plan descriptors already available with LwCCM.

Deployment-time binding of DDS implementation may ease application benchmarking. It is also possible that the DDS implementation used by the component application could be chosen at deployment time, rather than compile time. This enhancement will allow developers to evaluate the merits and performance characteristics of different DDS implementations rapidly.

Increased reliance on tooling. A consequence of developing with DDS4CIAO is the increased reliance on tooling, especially modeling tools. While writing the IDL and business logic for DDS-4CIAO components is straightforward, writing the deployment descriptors by hand is a difficult task that requires expert knowledge of the D&C specification. While the use of modeling tools — such as our CoSMIC toolsuite [9] or commercial tools that have emerged — can substantially ameliorate this concern, their use may not always be practical (CoSMIC, for example requires Windows while the commercial tools may be costly). A domain specific language (DSL) for describing deployments, configuration, and component packaging would substantially reduce the modeling requirement.

Applying connectors to the CCM CORBA infrastructure. The connector-based approach to integrating the DDS distribution middleware into CIAO has shown substantial promise. Unfortunately, however, the CORBA infrastructure that underlies CIAO/CCM still remains tightly integrated into the container implementation. A similar connector based approach could be used to convert Lw-CCM into a "Common Component Model", which is completely agnostic to the underlying communications middleware, by moving all of the extant CORBA communications functions to connectors themselves. This approach has the advantage of not only being able to remove the CORBA infrastructure currently used for synchronous two-way communication, but also makes it possible to, for example, swap in an alternative non-CORBA based connector implementation, if desired.

CIAO, DAnCE, and DDS4CIAO are available in open-source format from `download.dre.vanderbilt.edu`.

References

[1] Alejandro de Campos Ruiz and Gerardo Pardo-Castellote and GianPiero Napoli and Fernando Crespo-Sanchez and Javier Sanchez Monedero. High-level Programming of DDS Systems. In *Proceedings of the OMG Annual Real-time and Embedded Systems Workshop (RTWS)*, Arlington, VA, Mar. 2011.

[2] Angelo Corsaro. Simple API for DDS. `http://code.google.com/p/simd-cxx/`.

[3] L. Bulej and T. Bures. A connector model suitable for automatic generation of connectors. Technical report, 2003.

[4] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. *Software Engineering Research, Management and Applications, ACIS International Conference on*, 0:40–48, 2006.

[5] C. Esposito and D. Cotroneo. Resilient and timely event dissemination in publish/subscribe middleware. *International Journal of Adaptive, Resilient and Autonomic Systems*, 1:1 – 20, 2010.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[7] J. Hill, D. C. Schmidt, J. Slaby, and A. Porter. CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments. In *Proceedings of the 15th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)*, Belfast, Northern Ireland, Apr. 2008.

[8] Ke Jin. Component-Based CORBA+DDS Applications in PocoCapsule vs CCM. `http://www.pocomatic.com/docs/whitepapers/corba/`.

[9] T. Lu, E. Turkay, A. Gokhale, and D. C. Schmidt. CoSMIC: An MDA Tool suite for Application Deployment and Configuration. In *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA, Oct. 2003. ACM.

[10] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[11] Object Management Group. *Lightweight CORBA Component Model RFP*, realtime/02-11-27 edition, Nov. 2002.

[12] Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.2 edition, Jan. 2007.

[13] Object Management Group. *DDS for Lightweight CCM Version 1.0 Beta 2*. Object Management Group, OMG Document ptc/2009-10-25 edition, Oct. 2009.

[14] OMG. *Deployment and Configuration of Component-based Distributed Applications, v4.0*, Document formal/2006-04-02 edition, Apr. 2006.

[15] D. A. Wheeler. Sloccount, a set of tools for counting physical source lines of code, 2009.

[16] M. Xiong, J. Parsons, J. Edmondson, H. Nguyen, and D. C. Schmidt. Evaluating Technologies for Tactical Information Management in Net-Centric Systems. In *Proceedings of the Defense Transformation and Net-Centric Systems conference*, Orlando, Florida, Apr. 2007.